TRANSLATION VALIDATION OF DECOMPILATION

BY

SANDEEP DASGUPTA

Ph.D. PRELIMINARY EXAM

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Vikram S. Adve, Chair
Professor Grigore Rosu
Professor R. Sekar
Professor Tao Xie

Abstract

The ability to directly reason about the binary code is desirable, not only because, in many circumstances, it is the only way to prove (or disprove) properties of the code that is actually executed (for example, when the source code is not available in case of legacy code or malware), but also because it avoids the need to trust the correctness of compilers [Tho84, BR10].

Binary analysis is generally performed by existing decompiler projects [RD14, Rem18, SWS+16, BJAS11, Alv18], by (1) translating machine code to an intermediate representation (IR), and thereby exposing many high-level properties (like control flow, function boundary and prototype, variable and their type etc.) of the binary, which are otherwise lost during the compilation pipeline, and (2) performing the analysis at the IR level. Analyzing the binary using the abstractions lifted to such high-level IR assist further analysis and/or optimization. Even though such analyzes are agnostic to any specific high-level IR, but many projects [RD14, Rem18, FCD18, reo, llvb] prefer to employ LLVM IR. LLVM IR, being an industry standard compiler IR, enables many analyses and optimizations out-of-the-box which allows building a static binary analyzer with minimal effort.

Formally establishing faithfulness of the decompilation (i.e. translation from machine code to high level IR) is pivotal to gain trust in any binary analysis and, to the best of our knowledge, there is no exiting work which proves equivalence of a translation from binary to high-level IR. We believe an effort in the direction would enable both the developers of binary translators, to validate their implementation, and the clients of those translators, to gain trust in their analysis results.

The goal of our work is to formally validate such translation by leveraging the semantics of the languages involved (e.g. the Intel's X86-64 and LLVM IR). Some high level steps towards that goal involves defining the semantics of X86-64 and LLVM IR using K framework, a framework to define language semantics, and using the formal analysis tools which K provides "out-of-the-box" to develop an equivalence checker in order to prove equivalence between programs written in those languages. We have already taken the first big step towards the goal by formally defining the most complete X86-64 user-level ISA [DPK+19]. Moreover, the semantics of a subset of LLVM IR is already available [LLVa] for us to leverage and build on.

Given the recent advances in translation validation in validating compilation, employing translation validation to validate binary translators seems a promising direction to explore, however, there are additional challenges to deal with when it comes to validating the decompilation pipeline, like finding function, basic block and variable correspondence for

references are sorted alphabetically

sentence rephrased

checking equivalence, identifying and pruning glue code added during decompilation, to make the equivalence checker aware of the optimizations performed by the decompiler (e.g. recovering prototype and variables). Moreover, we would like validation approach to be general enough and hence applicable to any state-of-the-art binary to LLVM IR translators [RD14, Rem18, FCD18, reo, llvb], thereby avoiding any translator specific customization during validation. However, having this goal makes the problem even more challenging.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Analyzing binary code is crucial in software engineering and security research. Some of the notable applications of binary analysis can be found in binary instrumentation ([BGA03, LTCS10, LCM+05, NS03, Bru04]), binary translation [CE00], software hardening ([CWB15, FC08, ZS, ZWC+13]), software testing ([CKC11, ARCB14, GLM08]), CPU emulation ([Bel05, MCE+02]), malware detection ([CJS+05, KRV04, SBY+08, BJAS11, EKK+07, YSE+07]), automated reverse engineering ([CPC+08, LZ08, SLWB13, YEGPS15, RD14, Ang18, Alv18]), sand-boxing [KBA02, EAV+06, YSD+09], profiling [HM05, SE94], and automatic exploit generation [CARB12].

Binary analysis is generally performed by various decompiler projects [RD14, Rem18, SWS+16, BJAS11, Alv18], whose very first step is to translate the machine code to an intermediate representation (IR), and thereby exposing many high-level properties (like control flow, function boundary and prototype, variable and their type etc.) of the binary, which assist further analysis and/or optimization. Formally establishing faithfulness of the decompilation (i.e. translation from machine code to high level IR) is pivotal to gain trust in any binary analysis. Any bug in the translation would invalidate the binary analysis results. For example, a malware analysis system might miss vulnerabilities or a binary instrumentation system, instrumenting a buggy IR, might lead to failure or even crash in interpreting the instrumented program. Therefore, automatic validation tools are needed urgently to uncover hidden problems in a binary translator.

Despite of the importance in establishing the faithfulness of the binary translators, there has been surprisingly little effort towards that direction. The most notable approaches are either based on hardware co-simulation testing [MPRB09, MPFRB10], which is limited because generating specific test-cases to uncover semantic bugs in a lifter is non-trivial, or differential-testing [MMP+12, KFJ+17], which is limited in terms of coverage of the instructions validated and faithfulness guarantees it provides (Refer to Chapter 3).

We propose to employ translation validation on binary lifters, as a means to establish the faithfulness of the translation, by leveraging the semantics of the languages involved (e.g. the Intel's X86-64 and the high-level IR). Given the recent advances in translation validation [PSS98] in validating compilation [Nec00, PSS98, STL11, TGM11, ZPFG02, BFG+05], employing translation validation to validate binary translators seems a promising direction to explore, however, there are additional challenges to deal with when it comes to validating the decompilation pipeline (Refer Section 4.2). Moreover, we would like validation approach to be general enough and hence applicable to any state-of-the-art binary to high-level IR

translators [RD14, Rem18, FCD18, llvb, BJAS11, SWS+16, DFPA17]. The idea is to validate the translators without leveraging any knowledge of the translation involved, which in turn makes the problem even more challenging (Refer Chapter 6).

We believe that formally validating the translation is more robust as compared to (1) validating the translation using random (or low coverage) test-cases, and (2) differential testing technique which tests the correctness of a translation, generated by a translator, by comparing its behaviors with that provided by other translators under test.

Contributions Below we propose the primary contributions of our work.

Translation validation on binary translators We propose tools and techniques to formally validate the translation from binary to high level IR. To the best of our knowledge, we are the first to propose translation validation to establish the faithfulness of binary lifters targeting LLVM as their high-level IR. We would demonstrate the applicability of our approach by validating the translation of a realistic decompiler, McSema [RD14].

Most Complete user-level x86-64 Semantics: Towards the goals of translation validation of binary translators, we developed[DPK+19] the most complete and thoroughly tested formal semantics of x86-64 to date, which faithfully formalizes all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture.

Automated back-box approaches to translation validation We would like our technique to work automatically and uniformly across translators considering the translators as black-boxes, i.e. without using any translator-generated hints or translator-specific heuristics to assist the validation. We would like to demonstrate the effectiveness of our solution by validating against two or more decompilers. Optionally, we would like to use machine learning techniques to infer the variable or basic block correspondence between binary and LLVM IR code which will help in realizing a translator-agnostic approach to translation validation.

CHAPTER 2: BACKGROUND ON DECOMPILERS: FACILITATING BINARY ANALYSIS

Analysis and reasoning about source code is one of the most pervasive concepts in computer science research. Analyzing the code to approximate the semantics of the program helps in determining the correctness of the program w.r.t some gold standard or determining illegal memory and control flow accesses, or proving/refuting various properties of interest to the users. Static analysis [NNH10], model checking [CE82, QS82], and abstract interpretation [CC77] are the well known techniques used, widely and with ease, for the analysis of source code and have been deployed in many tools and processes that improve the software quality [XCE03, MQB+08, IYG+05, DHR+07, Bin07, BBC+10, BBC+06]. All such static source code analysis techniques are targeted to human readable source code written in high level languages as apposed to low level binary code. Analysis at the binary level is difficult mainly because many source level information, e.g. loops, procedures, or classes which provides a natural structural partitioning of programs into functionally related units are completely or partially lost during the compilation process. Moreover, some source level information, like symbol information, types, function boundaries and their prototypes, which creates a logical view of the program within a structural partition, is also stripped off during the compilation process. Absence of symbol information and types means that variables are not easily identified, but are represented by reusable registers and the memory, which is addressable as a large continuous array. Registers and memory carry no type information, and pointers of any type are indistinguishable from integers. Despite of these difficulties, there are several compelling reasons to do analysis at the binary level, We enumerate the most important ones as follows:

- Analyzing stripped binary executables, i.e., binaries without symbol or debugging information, enables software analysis without access to source code. Such scenarios arise in the case of (1) legacy code, when binary analysis is the only viable option to re-implement (or patch) the program, or (2) malwares, when binary analysis helps in security audits or malware detection [CJS+05, HKV07, KKSV05, KKSV10, KCK+09].

- While analyzing source code, the libraries are often replaced by coarse grained abstractions [GR07]. Operating on the binary avoids these issues altogether, since all source languages are translated into a hardware specific, but single target language with no distinction between the source code or library code.

- Even when the source code is available, doing analysis at the binary level alleviates the need to trust the correctness of the compiler. Moreover, during the compilation process,

the source code undergoes many modifications, removal or additions, before translated to binary and analyzing that binary is desirable because it is what is actually executed on hardware [BR10].

Binary analysis is not easy [MM16] and few long standing challenges can be enumerated as follows:

- Code and Data Ambiguity
- No Fixed Procedure Layout
- Missing or Untrusted Symbol Information
- Complex instruction Set
- Indirect Branches
- Overlapping Instructions
- Abusing Calls and Returns
- Lack of Types
- Presence of Non-returning functions

Despite the various challenges in analyzing machine code, there has been impressive amount of work to rebuild a close approximation of the high-level source code from a compiled binary using various decompilation frameworks [RD14, Rem18, SWS+16, BJAS11, Alv18, FCD18, SBY+08, hex, FDCT11, Eri18, KRS18, SLWB13, Gui08, SY18, CC11].

Binary analysis using a decompilation framework is achieved by (1) Translating machine code to an intermediate representation (IR), which precisely represents the operational semantics of the binary code. The lifted IR exposes many high-level properties (like control flow, function boundaries and prototypes, variables and their types etc.) of the binary, which are otherwise lost during the compilation pipeline, and (2) performing the analysis at the IR level. Analyzing the binary using the abstractions lifted to such high-level IR assists further analysis and/or optimization.

Binary analysis is mostly agnostic to any specific high-level IR, but many projects [RD14, Rem18, FCD18, reo, SY18] prefer to employ LLVM IR [LA04a]. LLVM IR, being an industry standard compiler IR, enables many analyses and optimizations out-of-the-box which allows building a static binary analyzer with minimal effort. For these reasons, we focus on decompilers to LLVM IR, but we believe that the core techniques are applicable to other decompilers targeting mid-level (language-neutral) IRs.

CHAPTER 3: RELATED WORK

Given the importance of establishing the faithfulness of the binary lifters, there exists a couple of efforts towards that direction, which we will elaborate on next.

3.1 RECENT ADVANCES IN VALIDATION OF DECOMPILATION

All the previous approaches can be broadly categorized to be based on (1) Simulation-testing, (2) Formal Methods, or (3) Machine Learning.

3.1.1 Approaches using Simulation-Testing

This approach is similar to black-box testing in software engineering. Most notable work include Martignoni et al. [MPRB09, MPFRB10, MMP+12] and Chen *et al.* [CYS+15].

Martignoni et al. [MPRB09, MPFRB10] proposes hardware-cosimulation based testing on QEMU [Bel05] and Bochs [Law96]. Specifically, they compared the state between actual CPU and IA-32 CPU emulator (under test) after executing randomly selected test-inputs on randomly chosen instructions to discover any semantic deviations. Although, a simple and scalable approach, it's effectiveness is limited because many semantics bugs in binary lifters are triggered upon a specific input and exercising all such corner inputs, using randomly generated test-cases, is impractical.

Chen *et al.* [CYS+15] proposed validating the static binary translator LLBT [SCHY12] and the hybrid binary translator [SYYH12], translating ARM programs to x86 programs using the LLVM x86 backend. First, an ARM program is translated offline to x86 program. Next, the translated x86 binary is executed directly on a x86 system while the original ARM binary runs on the QEMU emulator. During run time, both the ARM binary and the translated x86 binary produce a sequence of architecture states, which are compared at the granularity of single instruction or single basic block. They evaluate their validator using the ARM code compiled from EEMBC 1.1 benchmark.

Martignoni *et al.* [MMP+12] uses symbolic execution on a Hi-Fi emulator [Law96], defined as a binary emulator which is more complete in terms of instructions coverage of IA-32 ISA and faithful, to generate high-quality test cases to validate a Lo-Fi emulator [Bel05], defined as less complete and buggier emulator. The validation works as follows: A randomly chosen binary instruction is executed twice, once on a real hardware and next on the Lo-Fi emulator, and the output states are matched. Note that, even though Martignoni *et al.* [MMP+12]

symbolically explored the test-cases which is supposed to cover all the paths of a given instruction's implementation, but being a differential testing-based approach, the faithfulness depends directly on the faithfulness of the Hi-Fi emulator. A wrong implementation (or even omission of a particular case) of instruction semantics in the Hi-Fi, will lead to test-cases insufficient to explore all the paths and hence find bugs in the Low-Fi emulator[1]. Moreover, the symbolic execution of an instruction's implementation in the Hi-Fi emulator is achieved using an X86 interpreter FuzzBALL. A bug in the interpreter will affect the generation of high-fidelity test cases for a particular instruction, leading to incomplete coverage of that instruction's implementation in Low-Fi emulator. Also, the method can capture deviations in the behavior of only those instructions which are implemented in both the emulators.

3.1.2 Using Formal Methods

Another direction to establish strong guarantees in the binary translation is by using formal methods. The efforts along the direction include Soomin *et al.* [KFJ+17], Myreen et al. [MGS08, MGS12] and Frędęric *et al.* [RBB+19].

MeanDiff [KFJ+17] proposed an N-version IR testing to validate three binary lifters, BAP [BJAS11], BINSEC [BHL+11], and PyVEX [PYV] by comparing their translation of a single binary instruction to BIL, DBA, and VEX IRs respectively. The tools symbolically execute each of the IR instances, lifted from a single binary instruction, to generate symbolic summaries to be compared using a SAT solver. MeanDiff neither handle floating point operations, nor the instructions which does not manifest their side-effects (like flag updates) explicitly. Moreover, MeanDiff reports a bug whenever a deviation is detected w.r.t the instruction-semantics-behavior in at least two binary lifters. But even if all the binary lifters are in sync on the behavior of a particular instruction, we cannot guarantee that all the lifters are faithful in lifting that instruction, which is however, a general limitation of differential testing based approach. Also, as MeanDiff is testing multiple binary lifters together, hence it cannot be used to establish the faithfulness in lifting the semantics of an instruction which is not implemented in any one of them.

Myreen et al. [MGS08, MGS12] proposed "decompilation into logic" which, given some concrete machine code and a model of an ISA, extracts logic functions or symbolic summaries which captures the functional behavior of the machine code. The decompiler works on top of ISA models for IA-32 [CS03], ARM [Fox03] and PowerPC [Ler06]. Assuming that the ISA models are trusted, the extracted functions can be used to prove properties of the original

[1]We note that our proposed semantics-driven translation validation approach shares similar assumptions about the faithfulness of the semantics.

machine code. However, the work has not been applied to validate the binary translation to an IR. A recent work by Roessle et al. [RVR19] improves the aforementioned idea by including a subset of x86-64, derived mostly from Strata [HSSA16], in the trust-base of ISA models.

Leveraging symbolic summaries of source and target programs to prove the correctness of compilation/optimization is a well-researched topic. In the context of binary translation, MeanDiff [KFJ+17] has demonstrated how symbolic summaries can help in N-version equivalence testing of binary lifters. However, they compared the summaries corresponding to a single instruction at a time.

3.1.3 Using Machine Learning

Another recent work by Schulte *et al.* [Eri18] proposed Byte-Equivalent Decompilation (BED) which leveraged a genetic optimization algorithm to infer C source code from a binary. Given a target binary and an initial population of C code as decompilation candidates, they drive a genetic algorithm to improve the initial candidates, driving them closer (using compilation to binary) to byte-equivalence w.r.t the target binary. The byte equivalence is simply the edit distance to the target binary. As hypothesized in the future work section of the paper [Eri18], BED could be applied to LLVM IR instead of C to evolve lifting from machine code to LLVM IR and may work well due to the relative simplicity of LLVM IR as compared to the C. Being byte-equivalent, the generated LLVM IR will be the faithful evolution from the machine code. However, as shown in the paper, this approach worked moderately well for smallish binaries. For example, out of 22 binaries under test, only 4 achieve full byte equivalence when the initial population is augmented with decompilation candidates from the HEX-RAYS [hex] Decompiler. It is still an open problem to realized an end-end byte-equivalent binary to LLVM decompiler using purely genetic optimization algorithm.

Next, we elaborate on the state-of-the-art on various techniques and concepts which we propose to borrow as the basic ingredients to build our approach on. Those include Equivalence checking, Translation Validation, and Software Verification, and Machine Learning-Assisted Binary Code Analysis.

3.2 TRANSLATION VALIDATION

Pnueli *et al.* [PSS98] proposed the idea of translation validation as a new approach to the verification of translators (compilers, code generators). The idea is: Rather than verifying

the compiler itself, one constructs a validation tool which, after every run of the compiler, formally confirms that the target code produced in the run is a correct translation of the source program. One of the important ingredients to set up the translation validation process involve a formalization of the notion of "correct implementation" as a refinement relation. As proof method for the refinement, they employ a generalization of the well-established concept of simulation with refinement mapping [AL91]. Refinement mappings define a correspondence between the variables of a concrete system and the variables of an abstract system such that observations are preserved.

Hawblitzel et al. [HLP+13] use a translation validation approach to determine whether assembly code produced by different versions of the CLR JIT compiler are semantically equivalent and thus report mis-compilations when there are differences. The versions include those across a seven-month time period, across two architectures (x86 and ARM), and across optimizations levels. The underlying validator encodes each assembly method body into a procedure in the Boogie [BCD+06] programming language and then invokes the SymDiff symbolic differencing tool [LHKR12] to compare the Boogie encodings for semantic equivalence. For code with loops, the validator simply eliminates loops by unrolling them n (= 2) times, ignoring any behaviors past the n^{th} iteration.

Translation validation has been employed heavily in the field of compiler correctness [ZPFG02, BFG+05, Nec00]. Necula [Nec00] proposed a technique where each of the original and the optimized programs is firstly evaluated symbolically into a series of mutually recursive function definitions. A basic block and variable correspondence is inferred by a scanning algorithm that traverses the function definitions. The algorithm generates both a relation between program points and the accompanying constraints between program variables and memory at the program point.

For example, when the scanning algorithm visits a branch condition `e` in the original program, it determines whether `e` is eliminated due to the optimizations. If it is eliminated, then the information collected is either `e = 0` or $\sim$`e = 0`, depending on which branch of `e` is preserved in the optimized program. If `e` is not eliminated, then it corresponds to another branch condition `e'` in the optimized program. The information collected is either `e = e'` or `e =` $\sim$`e'`, depending on the correspondence of `e`'s and `e'`'s branches. One of the limitations of the algorithm is that all branches in the target program must correspond to branches in the source program. Moreover, to find a basic block correspondence Necula's technique uses some heuristics which are specific to the GNU C compiler.

The translation validation technique by Rival [Riv04] provides a unifying framework for the certification of compilation and of compiled programs. Similarly to Necula's technique, the framework is based on a symbolic representation of the semantics of the programs. Rival's

technique extracts basic block and variable correspondence from the standard debugging information if no optimizations are applied. However, when some optimizations are involved in the compilation, the optimizing phase has to be instrumented further to debug the optimized code and generate the correspondence between the original and the optimized programs. One technique to automatically generate such a correspondence is due to Jaramillo et. al [4]. In this technique, the optimized programs initially starts as an identical copy of the original one, so that the mapping starts as an identity. As each transformation is applied, the mapping is changed to reflect the effects of the transformation. Thus, in this technique, one needs to know what and in which order the transformations are applied by the optimizing phase.

3.3 MACHINE LEARNING-ASSISTED BINARY CODE ANALYSIS

Hasabnis *et al.* [HS16b, HS16a] presents the semantics of x86-64 using machine learning [HS16b] and symbolic execution [HS16a] to automatically learn the translation of x86-64 instructions to their IR, by extracting knowledge from the hard-coded translation logic of compilers such as GCC. However, as they admitted [HS16a], their semantics omits some important details of x86-64 semantics (e.g., the effect of various instructions on CPU flags).

Jaffe *et al.* [JLS+18] proposes a technique to assign meaningful variable names to Hex-ray [hex] decompiled C code by learning names that developers have assigned to code used in similar contexts. The technique aligns the variables in the decompiler output with those in the original C code in order to generate an aligned parallel corpus that is suitable for training a Statistical Machine Translation model [KHB+07].

Rosenblum *et al.* [RZMH07, RZMH08]consider the machine learning problem of identifying function entry points in binaries where symbols indicating function location are stripped. Rosenblum *et al.* [RMZ10] formulate compiler identification as a structured learning task, automatically building models that classify sequences of stripped binary code by the generating compiler. They train their model using binaries compiled using GCC and ICC and labeling each binary program point with a particular compiler.

Rosenblum *et al.* [RZM11] developed an authorship-attribution technique that uses machine learning approach to automatically discover the stylistic characteristics of binary code which are indicative of programmer style.

CHAPTER 4: OVERVIEW OF THE APPROACH

4.1 PROBLEM STATEMENT

Given the importance of binary analysis, establishing the faithfulness of binary lifter (or translator) is most desirable, which is what we propose as our future work. Having said that, we formulated our problem statement as follows:

Problem Statement: *Developing tools and techniques to formally validate the translation from binary to high level IR*

Specifically, we would like to validate the translation from x86-64 binary program to high level LLVM [LA04b] IR. This is in contrast with verifying the translator[1], as in compiler verification [Ler09], which is not only more involving but is specific to individual translators. Whereas, translation validation allows us to apply the technique to multiple state-of-art binary lifters/translators.

Next, we will discuss challenges in the proposed work (Section 4.2) and the detailed steps (Section 4.3) that we envision to mitigate those.

4.2 CHALLENGES IN TRANSLATION VALIDATION OF DECOMPILERS

According to the pioneering work by Pnueli *et al.* [PSS98], following are three key ingredients to set up a fully automatic translation validation system:

4.2.1 Providing Common Semantic framework

This ingredient requires a common semantic framework in order to represent both the source (as x86-64 binary program) and the target code (as LLVM IR). Such a common framework allows the source and target programs to be both represented using a common specification language which assist reasoning over them. Moreover, the employed semantics framework need to be general enough to model programs written in different languages.

One of the long standing challenge, which deters many previous efforts like [KFJ+17], in employing a formal validation approach for binary lifters, is the lack of formal specification

[1]Translator verification uses a theorem prover or proof assistant to verify a translator correct, once and for all, so that all translator output is guaranteed correct for all valid source programs. Translation validation, on the other hand, runs a theorem prover after each translation to check that the output of the translator is semantically equivalent to the source program.

of the x86-64 instructions specifying binary program. x86-64 instruction are massive in both numbers and complexity and are specified in an informal manner in the Intel manual [Int18] worth over $3,800$ pages. The x86-64 ISA is huge , partly because of a large number of complex instructions and partly because it keeps most of the legacy and deprecated instructions (336+) for the sake of backwards compatibility. It consists of 996 mnemonics, and each mnemonic admits several variants, depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands. Moreover, the x86-64 reference manual informally explains the instruction behaviors, leaving certain details unspecified or ambiguous, which requires to consult with an actual processor implementation to clarify such details. Completely formalizing the vast number of instructions with carefully identifying all the corner cases from the informal document, thus, is highly non-trivial.

4.2.2 Providing Formal Notion of Program Equivalence

This ingredient expects the existence of designated control location(s) in the computations of the source and target programs, that can serve as an observation point(s) for comparing the values of a set of externally observable variables (input/output variables, for example). The challenge here is to formally establish the notion that the lifted LLVM IR correctly implements the x86-64 binary code. Given the recent advances in translation validation in validating compilation [ZPFG02, BFG+05, Nec00, SMK13, KTL09, NZ13, SSCA13, TSTL09, TGM11], a basic version of this strategy is likely quite feasible today, however, there are additional challenges to deal with when it comes to validating the decompilation pipeline. For example, finding variable or basic block correspondence for checking equivalence is complicated by glue-code[2] added by the lifter during decompilation. Moreover, decompilers like McSema [RD14],llvm-mctoll [llvb] and reopt [reo] performs analysis and optimization on top of the decompiled IR to identity function prototypes, types and variables. Naive approaches, like identifying and/or pruning glue code added during decompilation or to make the equivalence checker aware of the optimizations performed by the decompiler, might work for a specific decompiler but, being ad-hoc, will not be easily scalable to others.

4.2.3 Providing automated proof generator and checker

This ingredient is about generating a machine-checkable proof of valid translation for a given pair of input and output programs as per the notion of equivalence established above. The proof can be generated by the binary-translator itself. Intuitively, the proofs, using an

[2]This is mainly to facilitate native execution of external function calls

equivalence checker, will try to establish equivalence between synchronization points which are symbolic descriptions that capture the set of pairs of relevant states of the input and output programs. One of the essential requirements of the equivalence checker, in the context of binary lifting, is language independence, to proof equivalence between programs in different languages (like x86-64 and LLVM).

4.3 PROPOSED APPROACH

In this section we would outline our setup to validate the translation from x86-64 program to LLVM IR and propose solution to mitigate some of the challenges mentioned in Section 4.2.

As the common semantic framework for the representation of the binary code and the lifted LLVM IR code, we decided to build on K [ŞAL$^+$]. Given a syntax and a semantics of a language, K generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional cost. Using the interpreter, one can test the semantics immediately, which significantly increases the efficiency of semantics developments. Furthermore, the formal analysis tools facilitate formal reasoning about the given language semantics. This helps in terms of both applicability of the semantics and engineering the semantics itself.

In formal-method's literature, the notion of program (or semantic) equivalence is usually formalized as a bi-simulation relation [San11] between pairs of states of the two programs. Ideally, we want a bi-simulation variant that allows us to simply relate the states where the two programs actually synchronize, i.e., at the start of corresponding functions or basic blocks, at the loop headers, etc. Such related states are called synchronization points, which are a pair of symbolic states of the input and output programs accompanied by a set of equality constraints over symbolic variables found in the two states. Kasampalis *et al.* [KPAR] realized such a variant of bi-simulation, referred to as *cut-bisimulation*, which is well-suited for translation across language changes, as it can ignore program points of no interest in the input and/or output program.

As for the last ingredient of a Translation Validation system, we would like to design proof generator and the checker with language-independence in mind. Language-independence makes them applicable for (1) different source (x86-64) and translated program (the high-level-IR), and (2) different high-level translated IRs (like LLVM, BIL etc.) supported by different binary lifters. Towards that goal, we proposed to borrow a language-independent equivalence checking algorithm `Keq` [KPAR] that can be parameterized with the input and output language semantics. The algorithm employs the notion of cut-bisimulation, mentioned above, and the synchronization point needed for the proof are provided as an input to the

algorithm. Keq is robust enough to apply to the Instruction Selection (ISel) phase of LLVM to validate the translation from LLVM IR to the output of the ISel phase, called Virtual x86.

The `Keq` tool, being parametric on the semantics of the input/output languages, need to be provided the semantics of LLVM and x86-64 as inputs. Given the size and the complexity of the x86-64 specification, providing the x86-64 semantics is a non-trivial task which deters many previous efforts from using semantics driven approach to validate the binary translation. We are the first to develop the most complete and thoroughly tested formal semantics of x86-64 to date (Refer Section 5). Our semantics faithfully formalizes all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture. This totals 3155 instruction variants, corresponding to 774 mnemonics. The semantics is fully executable and has been tested against more than 7,000 instruction-level test cases and the GCC torture test suite. This extensive testing paid off, revealing bugs in both the x86-64 reference manual and other existing semantics. Moreover, the semantics of a subset of LLVM IR is already available [LLVa] for us to leverage and build on.

CHAPTER 5: FORMAL SEMANTICS OF x86-64 USER-LEVEL ISA

The equivalence checker being parametric on the semantics of source (x86-64) and target (LLVM) languages of decompilation, one of the important steps towards the goal is to define these semantics. In this section, we will discuss our published contribution [DPK+19] is coming up with the most complete and thoroughly tested formal semantics of x86-64 to date.

5.1 CHALLENGES IN FORMALIZING x86-64

The x86-64 instruction set architecture (ISA) is one of the most complex and widely used ISAs on servers and desktops, and ensuring the correctness of the x86-64 binary code is important. Completely formalizing the semantics of x86-64, however, is challenging especially due to the complexity and the sheer number of instructions that are informally specified in approximately 3,000-page standard [Int18].

Size and Complexity: The x86-64 ISA has a large number of instructions, partly because of a large number of complex instructions and partly because it keeps most of the legacy and deprecated instructions ($\sim$ 336+) for the sake of backwards compatibility. It consists of 996 mnemonics, and each mnemonic admits several variants, depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands.

Inconsistent Instruction Variants: Some variants have divergent behaviors more than the difference of their type and size. For example, `movsd`, one of the 128-bit SSE instructions, has very different behaviors depending on whether the type of the source operand is register or memory; it clears the higher 64 bits of the target register only when the source type is memory. Indeed, we revealed bugs in other semantics due to their incorrect generalization of the variants' behavior.

Ambiguous Documentation: The x86-64 reference manual informally explains the instruction behaviors, leaving certain details unspecified or ambiguous, which required us to consult with an actual processor implementation to clarify such details. Completely formalizing the vast number of instructions with carefully identifying all the corner cases from the informal document, thus, is highly non-trivial.

Undefined Behaviors: The x86-64 standard also admits undefined behaviors that are implementation-dependent. Many instructions (32[1] out of 996 mnemonics) have undefined behaviors: their output values of the destination register or the %rflags register are undefined in certain cases. That is, the processor is free to choose any behavior in the undefined case.

Many existing semantics, however, simply "define" the undefined behaviors by following a specific behavior taken by a processor implementation. This approach is problematic because they do not capture all possible behaviors of different processor implementations. Indeed, we found discrepancies between existing semantics in specifying the undefined behaviors, where different semantics are valid only for different groups of processors. That is, such semantics are not adequate to formally reason about universal properties (e.g., portability) of a program that need to be satisfied for all standard-conforming processors. For example, the parity flag %pf is undefined in the logical-and-not instruction `andn`, where the processor implementation is allowed to either update the flag value (to 0 or 1), or keep the previous value. We found, e.g., that Remill [Rem18] updates the flag with 0, whereas Radare [Alv18] keeps it unmodified. Identifying and faithfully specifying all the undefined behaviors, thus, are desirable but challenging.

In our semantics, we faithfully modeled the undefined value as a unique symbol (called undef) whose value is nondeterministically decided each time within the proper range. These nondeterministic values are enough to capture and formally reason about all possible behaviors of the instructions for different processors (and even any future, standard-conforming processor).

5.2 EXISTING SEMANTICS FOR x86-64

To date, to the best of our knowledge, despite several explicit attempts [HSSA16, GHKG14, GHK17] and other related systems [Ler09, Rem18, LR13, HS16b, HS16a], no *complete* formal semantics of x86-64 exists that can be used for formal reasoning about x86 binary programs. Heule *et al.* [HSSA16] presented a formal semantics of x86-64, but it covers only a fragment (∼47%) of all instructions; as the authors of [HSSA16] candidly admitted, their synthesis methodology proved insufficient to add the remaining instructions primarily due to limitations of the underlying synthesis engine. Moreover, their semantics misses certain essential details [DPK+19]. Goel *et al.* [GHKG14, GHK17], on the other hand, specified a formal semantics in the ACL2 proof assistant [KMM00], allowing to reason about functional correctness, but their semantics covers only a small fragment (∼33%) of all user-

[1]These numbers are obtained by parsing the official manual "Volume 2: Instruction Set Reference" and cross checked with projects [SSA13, Fel18] investing similar efforts.

level instructions. There also have been several attempts [SWS+16, BJAS11, Alv18, HS16a] to *indirectly* describe the x86-64 semantics, where they define an intermediate language (IL), specify the IL semantics, and translate x86-64 to the IL. This indirect semantics, however, may not be general enough to be used for different types of formal analyses, since the IL might be designed with specific purposes in mind, not to mention that the translation may miss certain important details of the instruction behaviors.

5.3 OVERVIEW OF THE APPROACH

Briefly, our approach is as follows. We first defined the machine configuration and underlying infrastructure in the $\mathbb{K}$ framework, in order to define, execute and test the x86-64 semantics. To leverage previous work as much as possible, we took the semantic rules of all the instructions supported in Strata [HSSA16], which amounts to about 60% of the instructions in scope, in the form of SMT formulas. We corrected, improved or simplified many of the baseline rules. We then translated these SMT formulas from Strata into $\mathbb{K}$ rules using a script, and tested the resulting rules by comparing with the Strata rules using Z3. These steps give us a validated initial set of semantic rules in $\mathbb{K}$ for about 60% of the target instructions (our "baseline" set).

We attempted to extend the stratification approach in Strata to define additional rules automatically, in two ways: (i) augmenting their base set B, and (ii) constraining the search space manually using knowledge of instruction behaviors. Both these attempts failed; they worked only for a few instructions, but in general, we found them to be impractical. Specifically, we added 58 base instructions to the base set, and learned the semantics of 70 new instructions, which are variants of the added instructions, in 20 minutes, but no more even after we kept running for two days. Also, we tried constraining the search space by manually populating it with relevant instructions. The lesson we learned from these experiments is, getting the right set of base instructions or a constrained search space for a complex instruction need an insight about the semantics of that instruction itself. We found that the effort to extract such information from the manual is about the same as manually defining that instruction.

We then manually added $\mathbb{K}$ rules for the remaining 40% of the target instructions by systematically translating their description of the Intel manual into $\mathbb{K}$ rules, in some cases cross-referencing against semantics available in Stoke. The outcome was a complete formal specification of user-level x86-64 in $\mathbb{K}$.

We validated this semantics in three ways: First, we use the $\mathbb{K}$ interpreter to execute the semantics of *each* instruction for 7,000+ test inputs (each input is a processor state

configuration) and compared the output directly with the hardware behavior for the same instruction. Second, we repeated this experiment using the applicable programs in the GCC C-torture tests [CTO18]. Third, we compared against the semantics defined in the Stoke project for about 330 instructions that were omitted in Strata (thus not included in our baseline), using an SMT solver.

These validation experiments uncovered bugs [Bug18a, Bug18c, Bug18b] in the Intel manual, in Strata's simplification rules, and in the Stoke semantics. All these bugs were reported to the respective authors, and most have been acknowledged and some have been fixed. Following is the brief summary of inconsistencies we found during the validation experiments.

Inconsistencies Found in the Intel Manual According to the manual, the semantics of `vpsravd %xmm3, %xmm2, %xmm1` seems to depend on the lower 100 bits of %xmm3, whereas the actual hardware execution suggests that it should depend on the lower 128 bits. Similar inconsistencies are found in instructions with mnemonics `vpsllvd`, `vpsllvq`, `vpsravd`. Also, we found misleading typos related to instructions with opcodes `vpsravw`, `vpsravd`, `vpsravq`, `packsswb`. All these findings were reported and acknowledged by Intel as issues in the manual [Bug18a].

Inconsistencies Found in Strata's Simplification Rules While testing the instructions specifications borrowed from Strata, we found inconsistent behaviors with the actual hardware. Moreover, the inconsistencies were discovered in the formulas of floating-point instructions. This is not surprising because Strata models the floating-point instructions as uninterpreted functions which cannot be executed or tested on hardware. Their semantics are executable in our definition though, and thus we were able to test them thoroughly. Note that Strata generates the formulas for these instructions by symbolically executing the corresponding learned instruction sequences followed by a formula simplification pass. Therefore, errors in those formulas can be due to bugs either in the symbolic execution engine or in the simplification stage. Our testing shows that the second is true with the following evidence. The simplification rule add_double(A, 0) == A does not hold for $A = -0.0$. Same for add_single. These were reported [PC118]. Also, the simplification rule sub_double(A, A) == 0 does not hold for $A = NaN$. Same is true for sub_single. We found this bug in the branch of Stoke which is used in Strata. But this has been already fixed in the latest Stoke branch.

Inconsistencies Found in Stoke First, for instructions like `addsubpd %xmm1, %xmm2` , the order of addition and subtraction specified by Stoke is opposite to the one specified in the Intel Manual. Same is true with the mnemonic `addsubps`. (Found in 12 instruction variants.)

Second, the instruction `pslld %xmm1, %xmm2` implements a logical left shift of packed data by a count specified in %xmm1. Stoke's specification vectorized the operand %xmm1 which is incorrect according to the manual. Similar issues were found in instructions implementing the logical right shift operations on packed data. (Found in 18 instructions.)

Third, `cvtsi2sdl %eax, %xmm1` and `vcvtsi2sdl %eax, %xmm0, %xmm1` are respectively SSE- and AVX-versions of the instruction to convert a double-word (32-bit) integer to a scalar single-precision floating-point value. According to the manual, in the AVX-version, the destination bits $127-64$ of the register %xmm1 are updated to the corresponding bits in the first source operand %xmm0. This is in contrast to the SSE-version of the instruction where the destination bits $127-64$ should remain unmodified. Stoke specifies the semantics of the AVX-version similar to the SSE-version, which is incorrect. (Found in 4 instruction variants.)

Finally, some instructions, like `imulb %al`, which drive flag registers to an undefined state are not modeled correctly in Stoke. (found in 8 instruction variants)

All these errors were reported and confirmed [Bug18c, Bug18b].

Moreover, we illustrate the potential of our semantics to be used for formal analyses such as deductive program verification and program equivalence checking. For example, following are the three applications that we demonstrated:

Program Verification To demonstrate that our semantics can be used to verify x86-64 programs, we use the x86-64 verifier to prove the functional correctness of the sum-to-n program. The functional correctness property, which is $return-value = \sum_1^N n = N(N+1)/2$ is specified as reachability specifications, essentially a pair of pre- and post-conditions for the function computing the sum-to-n.

Symbolic Execution to find Security Vulnerability $\mathbb{K}$ automatically derives a correct-by-construction symbolic execution engine from the given semantics. Being instantiated with our semantics, the engine can be used to symbolically execute and explore all possible paths in the given x86-64 program. We demonstrated how this capability can be used to find a security vulnerability. We took a known security vulnerability using a code snippet of the HiStar [ZBWKM06] kernel. Using symbolic execution, we derive a path condition which can trigger the vulnerability along with the triggering inputs.

Translation Validation of Optimizations $\mathbb{K}$ also provides a program equivalence checker that can be used for the translation validation of compiler optimizations. We derived an x86-64 program equivalence checker from our semantics and used it to validate different optimizations of `popcnt` program which counts the number of set bits in the given input. We compiled the program with different optimizations: the GCC compiler optimizations (-O0, -O1, -O2, and -O3), and the Stoke super-optimization. We validated these optimizations by checking the equivalence between the optimized programs.

The $\mathbb{K}$ framework also enables one to represent our semantics as SMT theories, which allows others to easily reuse our semantics for their own purposes. Indeed, we have translated our semantics to Stoke [Das19] which can serve as a drop-in replacement of Heule *et al.*'s semantics [HSSA16] and can immediately benefit tools built on Stoke (e.g., [RVR19]).

Our formal semantics is publicly available at [Das18].

CHAPTER 6: ONGOING AND FUTURE WORK: TRANSLATOR-AGNOSTIC TRANSLATION VALIDATION

The entire chapter is revised

With the overall goal of establishing faithfulness of binary lifters, we propose to use translation validation to validate the translation from binary to LLVM IR[1]. As mentioned earlier, we employed $\mathbb{K}$ framework as the formalism medium to represent the source and target language and in the process, defined the most complete semantics of user-level x86-64 using the framework. We proposed to borrow a language-independent equivalence checking algorithm `Keq` [KPAR] that can be parameterized with the input and output language semantics.

As mentioned in Section 4.3, the notion of equivalence between binary program and the lifted LLVM IR is formalized as a bi-simulation relation [San11] between pairs of states of the two programs. These program points, also referred to as synchronization points, are generated by a proof generator and fed to the equivalence checker as inputs. There are various ways a proof generator can generate these synchronization points each having different challenges and trade-offs.

6.1 TRANSLATOR-SPECIFIC APPROACHES TO SYNCHRONIZATION POINT GENERATION

One approach to generate the synchronization points is by instrumenting the translator itself. The translator, being aware of which high-level IR instructions and control points are emitted for each binary instruction, can accurately generate the synchronization points. Moreover, each synchronization point is labeled with invariants over symbolic variables, corresponding to input/output program states, in order to cover only observably equivalent states. For each synchronization point, the equivalence-checker checks equivalence by delegating proof obligations to an SMT solver. Failure of a proof obligation can be attributed to either a bug in the translator or the fact that the source and target programs are not equivalent in the first place. We can rule out the second possibility because the source and target programs are not selected arbitrarily but they are the actual input and output of a translator, under test, and hence claimed to be equivalent by the translator itself.

However, the above approach requires modifications of the translator to generate the synchronization point points and hence it may not be easily scalable to other decompilers.

[1]However, we believe that the core techniques are applicable to other decompilers targeting mid-level (language-neutral) IRs

Another approach is to analyze the source (x86-64) code and target (LLVM IR) code of a particular translation and infer the synchronization points. We note that the analyzer has to take into account translators specific idioms while analyzing the target code because different binary translators might use different ways to lift the binary code in the target IR. We would demonstrate the applicability of this approach by validating the translation of a realistic decompiler like McSema [RD14]

6.2 TRANSLATOR-AGNOSTIC APPROACHES TO SYNCHRONIZATION POINT GENERATION

The fact that the proof generator is using translator generated hints or idioms used by the translator to generate the synchronization points & the associated invariants, makes it translator specific and hence not applicable uniformly across translators. We aim for a binary-translator-agnostic proof generator, which, indeed, makes the problem even more challenging. We would like to explore the following approaches towards that goal.

Data-driven synchronization point generation This approach is based on inferring the synchronization points using test-runs. There are two notables [NV05, SSCA13] along this direction which deals with checking equivalence between GCC [GCC] RTLs or x86 loops respectively.

Iman *et al.* [NV05] proposed an algorithm for solving the problem of finding basic block and variable correspondence between two GCC RTL programs compiled from the same source, but using different optimizations. The essence of their technique is interpretation of the two programs on random inputs and comparing the histories of value changes for variables. If the sequences of value changes of two variables are the same, then the variables probably correspond to each other, and the block in which the changes occur might also correspond to each other. However, as the paper do not consider generation of verification invariants at each point of correspondence, hence it is difficult to judge its effectiveness in proving the equivalence of the candidate RTL programs.

Rahul *et al.* [SSCA13] presented "data-driven equivalence checker" (DDEC) to check equivalence of two x86 loops which uses data-driven inference in order to guess candidate simulation relations[2]. Their approach does not assume any knowledge about the optimization performed while determining those simulation relations. The candidate simulation relations are checked using an SMT solver to determine equivalence of the loops. The cutpoints are

[2]A simulation relation consist of pair of program cutpoints, which breaks two loops into a set of pairs of loop-free code fragments and linear equalities as invariants to hold at those cutpoints.

chosen where the corresponding memory states agree on the largest number of values. The equality conditions at cutpoint is determined by using standard linear algebra techniques over the values of live variables recorded by inserting instrumentation. Moreover, DDEC is sound; a lack of test coverage may cause equivalence checking to fail, but it cannot result in the unsound conclusion that two loops are equivalent when they are not.

Our proposed approach is based on the insight that a sound guess of synchronization points do not result in proving two programs equivalent when they are not (false positives). Indeed, they might produce false negatives. Having said that, given x86-64 and translated LLVM programs, we might compile the LLVM program to binary and use the data-driven approach [SSCA13], mentioned above, to extract guesses of synchronization points. However, we do not want to add the compilation in our trust base and hence we can map the synchronization points back to the IR level and employ the equivalence checker `Keq` to proof the equivalence of the resulting LLVM IR and the original binary using the synchronization point guesses. However, these synchronization point, being data-driven guesses, may lead to failing the proof obligations for reasons other than a translation error. For example, the proof might fail if the spurious invariants are generated or the synchronization points are not sufficient, and it is highly non-trivial, but an interesting research problem, to automatically attribute the failure reasons.

There is a trade-off between translator-specific and translator-agnostic proof generator approaches. The synchronization points generated by the former is accurate, leading to trivial and automatic attribution of proof failure reason. On the other hand, the synchronization point guesses produced by the translator-agnostic proof generator has issues with automatic attribution of the failure reasons. We envision opportunities to exploit this trade-off by pruning off synchronization point guesses which a translator specific proof generator will never generate.

For example, the equivalence checker `Keq` [KPAR] is used for validating the compilation pipeline and it uses compiler generated hints to prove equivalence. The underlying algorithm mandates three broad categories of synchronization points which are necessary and sufficient to prove equivalence, viz., entry points, exiting points (which also include points before callsites), and the rest of the points (including loop entry points and points after callsites). We may use this information to prune redundant synchronization point guesses generated by a translator-agnostic proof generator like DDEC [SSCA13].

Learning synchronization points As discussed in Section 3.3, there exist many recent efforts [JLS+18, RZMH07, RZMH08, RMZ10, RZM11, BBW+14, SSM15] in extracting vari-

ous features (like meaningful variable names, compiler provenance, function entry points) using machine learning techniques which are useful for binary analysis. Most notable is the novel technique presented by Katz *et al.* [KRS18] for decompiling binary code snippets using a model based on Recurrent Neural Networks. The model learns properties and patterns that occur in source code and uses them to produce decompilation output.

The employed model is a sequence-to-sequence model based on RNNs. The model composed of a two recurrent neural networks: an encoder, which processes the input binary sequence, and a decoder, which creates outputs, the decompiled C code. Further, the sequence-to-sequence model contains a hidden state, which summarizes the entire input sequence. The training input contains of a large pair of strings representing the snippet of source C code and the corresponding binary output, derived from open source projects. The snippets are preprocessed by tokenizing each string and translating the resulting token list to a list of integers. Preprocessing also produces a dictionary to map which integers correspond to which tokens in both the higher-level and binary snippets. The encoder layer of the RNN is trained using the list of integers, derived from binary and the decoder layer is trained using the corresponding list of integers, derived from C code. During training, the encoder-decoder modifies its internal state and does not provide additional output. For testing, the test-binary snippets are translated into lists of integers, which is then fed to the trained encoder-decoder model to output a list of integers, which is later turned into a predicted higher-level C translation using the dictionary generated in the preprocessing step. There evaluation results suggest that the technique can be used to obtain a high level syntactic structure of the code.

For an input binary and the high-level IR, translated by an arbitrary decompiler, our goal is to learn the corresponding synchronization point. Using a straightforward supervised-learning approach, we could frame the problem as: To predict if an arbitrary pair of program points is a synchronization point or not? Such a supervised-learning formulation requires providing many input code snippets, correctly labeled with synchronization points, as the training set in the first place. Indeed, it is interesting to explore how to harvest the input corpus.

Another way of inferring the synchronization points is by using "Trial and Error", for which Reinforcement Learning [SB98] seems to be a better choice to explore. For the "Trial and Error" approach to work, there must be an efficient way to find the error state, namely, the tried synchronization point is non-valid. With that, the goal of inferring all the corresponding synchronization points of an input output program pair can be defined solely in terms of some *reward* given on each selection. For example, for each selection of synchronization point, a *reward* can be assigned using the number of values on which the corresponding memory states agree on, when they are both tested using a random set of same inputs.

We believe such learning techniques are promising for this problem and we would like to explore more on this in future.

REFERENCES

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[Alv18] Sergi Alvarez. Radare2. `https://rada.re/r/`, July 2018. Last accessed: April 25, 2019.

[Ang18] Angr: A powerful and user-friendly binary analysis platform! `http://angr.io/`, July 2018. Last accessed: April 25, 2019.

[ARCB14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 73–85, New York, NY, USA, 2006. ACM.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[BBW+14] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BFG+05] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. Tvoc: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 291–295, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[BHL+11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 165–170, Berlin, Heidelberg, 2011. Springer-Verlag.

[Bin07] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[BR10] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[Bru04] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[Bug18a] Bug reported in Intel Developer Zone: Possible errors in instruction semantics. `https://software.intel.com/en-us/forums/intel-isa-extensions/topic/773342`, April 2018. Last accessed: April 25, 2019.

[Bug18b] Bug reported in Stoke: Modelling the behavior of flags which may or must take undef values. `https://github.com/StanfordPL/stoke/issues/986`, May 2018. Last accessed: April 25, 2019.

[Bug18c] Bug reported in Stoke: Semantic bugs. `https://github.com/StanfordPL/stoke/issues/983`, April 2018. Last accessed: April 25, 2019.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CC11] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with revgen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 211–216, June 2011.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag.

[CE00] Cristina Cifuentes and Mike Van Emmerik. Uqbt: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, March 2000.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[CPC+08] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM.

[CS03] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In Franz Baader, editor, *Automated Deduction – CADE-19*, pages 106–120, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CTO18] C Language Testsuites: C-torture version 8.1.0. `https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`, 2018. Last accessed: April 25, 2019.

[CWB15] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 725–741, Washington, DC, USA, 2015. IEEE Computer Society.

[CYS+15] Jiunn-Yeu Chen, Wuu Yang, Bor-Yeh Shen, Yuan-Jia Li, and Wei-Chung Hsu. Automatic validation for binary translation. *Comput. Lang. Syst. Struct.*, 43(C):96–115, October 2015.

[Das18] Sandeep Dasgupta. Semantics of x86-64 in K. `https://github.com/kframework/X86-64-semantics`, 2018. Last accessed: April 25, 2019.

[Das19] Sandeep Dasgupta. Defining semantics of instructions unsupported in Strata/Stoke. `https://github.com/StanfordPL/stoke/pull/996`, 2019. Last accessed: April 25, 2019.

[DFPA17] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. ACM.

[DHR+07] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *2007 Future of Software Engineering*, FOSE '07, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[DPK+19] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[EKK+07] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[Eri18] Alexey Loginov Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta. Evolving Exact Decompilation. In *Binary Analysis Research (BAR), 2018*, number February, 2018.

[FC08] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[FCD18] fcd: An optimizing decompiler. `https://zneak.github.io/fcd/`, July 2018. Last accessed: April 25, 2019.

[FDCT11] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: Approaching c++ decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 347–356, Washington, DC, USA, 2011. IEEE Computer Society.

[Fel18] x86 and amd64 Instruction Reference (UnOfficial). `http://www.felixcloutier.com/x86/`, July 2018. Last accessed: April 25, 2019.

[Fox03] Anthony Fox. Formal specification and verification of arm6. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, pages 25–40, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[GCC] Gcc, the gnu compiler collection. `https://gcc.gnu.org/`. Last accessed: April 25, 2019.

[GHK17] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*, pages 173–209. Springer International Publishing, Cham, 2017.

[GHKG14] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 18:91–18:98, Austin, TX, 2014. FMCAD Inc.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS (Network and Distributed Systems Security)*, pages 151–166, 2008.

[GR07] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 68–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Gui08] Ilfak Guilfanov. Decompilers and beyond. In Black-Hat USA, July 2008.

[hex] Hex-Rays Decompiler. `http://www.hex-rays.com/decompiler.shtml`. Last accessed: April 25, 2019.

[HKV07] Andreas Holzer, Johannes Kinder, and Helmut Veith. Using verification technology to specify and detect malware. In Roberto Moreno Díaz, Franz Pichler, and Alexis Quesada Arencibia, editors, *Computer Aided Systems Theory – EUROCAST 2007*, pages 497–504, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[HLP+13] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 191–201, New York, NY, USA, 2013. ACM.

[HM05] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, December 2005.

[HS16a] Niranjan Hasabnis and R. Sekar. Extracting instruction semantics via symbolic execution of code generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 301–313, New York, NY, USA, 2016. ACM.

[HS16b] Niranjan Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 311–324, New York, NY, USA, 2016. ACM.

[HSSA16] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 237–250, New York, NY, USA, 2016. ACM.

[Int18] Intel 64 and IA-32 Architectures Software Developer Manuals. `https://software.intel.com/en-us/articles/intel-sdm`, 2018. Published on October 12, 2016, updated May 18, 2018.

[IYG+05] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 301–306, Berlin, Heidelberg, 2005. Springer-Verlag.

[JLS+18] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 20–30, New York, NY, USA, 2018. ACM.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[KCK+09] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[KFJ+17] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 353–364, Piscataway, NJ, USA, 2017. IEEE Press.

[KHB+07] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration*

Sessions, ACL '07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

[KKSV05] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'05, pages 174–187, Berlin, Heidelberg, 2005. Springer-Verlag.

[KKSV10] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Proactive detection of computer worms using model checking. *IEEE Trans. Dependable Secur. Comput.*, 7(4):424–438, October 2010.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[KPAR] Theodoros Kasampalis, Daejun Park, Vikram S. Adve, and Grigore Roşu. Cross-Language Program Equivalence with Application to LLVM. `https://drive.google.com/open?id=1OWvaQHMC5KFVnyp2motlDHYLauI7Luv0`. Under submission in SAS 2019: 26th Static Analysis Symposium.

[KRS18] Deborah Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *Software Analysis, Evolution and Reengineering (SANER), 2018*. IEEE, 2018.

[KRV04] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 327–337, New York, NY, USA, 2009. ACM.

[LA04a] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LA04b] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Law96] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es), September 1996.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[LHKR12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[LLVa] Formal semantics of llvm ir in k. `https://github.com/kframework/llvm-semantics`. Last accessed: April 25, 2019.

[llvb] llvm-mctoll. `https://github.com/Microsoft/llvm-mctoll`. Last accessed: April 25, 2019.

[LR13] Junghee Lim and Thomas Reps. TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4:1–4:59, April 2013.

[LTCS10] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 175–183, March 2010.

[LZ08] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 83–93, New York, NY, USA, 2008. ACM.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[MGS08] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures: An application of decompilation into logic. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 20:1–20:8, Piscataway, NJ, USA, 2008. IEEE Press.

[MGS12] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic — improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81, Oct 2012.

[MM16] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[MMP+12] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM.

[MPFRB10] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 171–182, New York, NY, USA, 2010. ACM.

[MPRB09] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 261–272, New York, NY, USA, 2009. ACM.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[NV05] Iman Narasamdya and Andrei Voronkov. Finding basic block and variable correspondence. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 251–267, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[NZ13] Kedar S. Namjoshi and Lenore D. Zuck. Witnessing program transformations. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 304–323, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[PC118] Eric Schkufza. Personal communication, June 2018.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[PYV] Python bindings for valgrind's vex ir. `https://github.com/angr/pyvex`. Last accessed: April 25, 2019.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[RBB+19] Fręd̨eric Recoules, Sębastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. Get rid of inline assembly through trustable verification-oriented lifting. `https://arxiv.org/abs/1903.06407`, 2019. Last accessed: April 25, 2019.

[RD14] Andrew Ruef and Artem Dinaburg. Static Translation of X86 Instruction Semantics to LLVM with McSema. *REcon*, 2014.

[Rem18] Remill: Library for lifting of x86, amd64, and aarch64 machine code to llvm bitcode. `https://github.com/trailofbits/remill`, July 2018. Last accessed: April 25, 2019.

[reo] reopt: A tool for analyzing x86-64 binaries. `https://github.com/GaloisInc/reopt`. Last accessed: April 25, 2019.

[Riv04] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 1–13, New York, NY, USA, 2004. ACM.

[RMZ10] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 21–28, New York, NY, USA, 2010. ACM.

[RVR19] Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 181–195, New York, NY, USA, 2019. ACM.

[RZM11] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, pages 172–189, Berlin, Heidelberg, 2011. Springer-Verlag.

[RZMH07] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Machine learning-assisted binary code analysis. `http://pages.cs.wisc.edu/~jerryzhu/pub/nips07-abs.pdf`, 2007. Last accessed: April 25, 2019.

[RZMH08] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[ŞAL+] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The K primer (version 3.2). Technical report.

[San11] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[SCHY12] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. Llbt: An llvm-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 51–60, New York, NY, USA, 2012. ACM.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[SLWB13] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22Nd USENIX Conference on*

Security, SEC'13, pages 353–368, Berkeley, CA, USA, 2013. USENIX Association.

[SMK13] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, 2013. ACM.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM.

[SSCA13] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[SSM15] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 611–626, Berkeley, CA, USA, 2015. USENIX Association.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[SY18] Aaron Smith and S. Bharadwaj Yadavalli. LLVM Based Binary Raiser: llvm-mctoll. 2018.

[SYYH12] B. Shen, J. You, W. Yang, and W. Hsu. An llvm-based hybrid binary translation system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 229–236, June 2012.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, August 1984.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 327–336, New York, NY, USA, 2003. ACM.

[YEGPS15] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[YSE+07] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[ZPFG02] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 65(2):2 – 18, 2002. COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002).

[ZS] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries.

[ZWC+13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.